AIAA 2000-4500

DATA MANAGEMENT IN THE ASSET SIMULATION FRAMEWORK

Jeffrey Maddalon[*], **Stephen Derry**[*]
{j.m.maddalon | s.d.derry}@larc.nasa.gov

NASA Langley Research Center
Hampton VA, 23681

Abstract

The Aircraft System Simulation Environment and Toolkit (ASSET) is a software framework for rapid development and simulation of continuous systems. The central concept of ASSET is a model—analogous to a block in a system block diagram. One important way to simplify the software development process for models is to use *encapsulation*. Encapsulation is a software development technique that disallows one model from accessing another model's internal state; models can only communicate through their well-defined input/output interface. Unfortunately when building a simulation, the engineer will often require visibility to data values that are not part of the normal interface. The data management system of the ASSET framework attempts to solve this problem. The data management system consists of a large central repository in which all models may write data and a common set of tools to access the data.

Introduction

The Aircraft System Simulation Environment and Toolkit (ASSET) is a software framework for rapid development and simulation of continuous systems. The framework was designed primarily to simulate aircraft and aircraft systems. ASSET is entirely written in the Java computer language from Sun Microsystems [1]. The Java computer language offers advantages over other languages in the areas of portability, support for the object-oriented design paradigm, and ease of learning.

[*] *Computer Engineer, Systems Development Branch*

The central concept of ASSET is a model [2]—analogous to a block in a system block diagram. Models have well-defined input and output vectors and may contain states and/or other models. A system can be simulated in ASSET with one or more models. Other parts of ASSET include a central data repository, software to present simulation data in a useful format and a graphical user interface to ease user interaction. In an environment with rapidly changing requirements, models often need to change frequently. To reduce the impact of this, one important goal of ASSET is to minimize the work of the model designer.

One method to achieve the goal of decreasing the work of the model designer is to protect a model from other models. The model designer must define the input and output vectors for the model, and these vectors are the *only* interface between the model and the world outside. Model designers do not need to concern themselves about how or when other models will read their data since models within ASSET can only access the well-defined interface of the model. The technique of isolating models from each other is termed *encapsulation* [3].

Software frameworks that do not implement encapsulation usually encounter difficulty maintaining scalability. As the system grows and more models are added, it becomes much more likely that one component will supply data to another component incorrectly. Models with a well-defined interface (that is, models with well-defined input and output vectors) should also be more reusable than a model without encapsulation. If someone wants to use an encapsulated model in another context, then all they need to provide are the input signals. Since models are guaranteed to be independent of each other, they should be able to be moved to many environments (such as batch or real-time). Finally, a well-defined interface allows model writers the freedom to change the implementation of a model without forcing the users of this model to change their software.

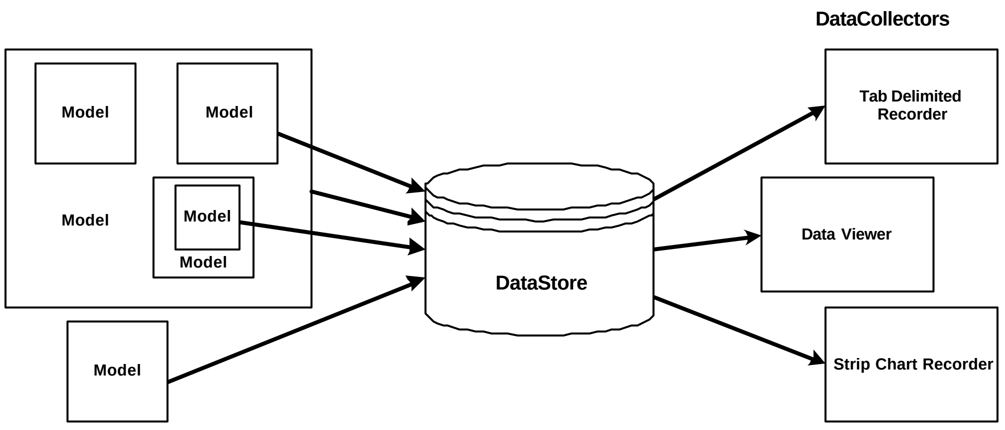

Figure 1 - ASSET Data Management System

To effectively implement encapsulation, the model designer should attempt to minimize the number of both inputs and outputs. Building a model with too many input signals brings all the problems of lack of encapsulation within the model and unnecessarily increases complexity on the model's owner. A model with too many outputs unnecessarily exposes the model's implementation and reduces its reusability.

This encapsulation technique provides powerful leverage for the ultimate scalability and reusability of the ASSET framework; however it comes at a cost. Often when developing a system, the designer wants to know not only the values of the input and output vectors but also the value of intermediate computations and states for any model in the system. These values may not be available through the output vector. The designer could add "test points" to the output vector; however, to maintain backward compatibility, every future implementation of this model must produce these signals—even if they are not relevant to the current implementation. In addition every time the designer wants another signal he or she must change the code and add this signal and recompile. Finally, the designer must actually have the model's source code and even if it were available, the designer would need a detailed understanding of the implementation to know where to place the test point. Is there a better way?

In the above scenario, one may recognize two distinct roles: the system designer who is concerned with simulating the whole system and the model designer who is concerned with one particular subsystem. Oftentimes, the same person fills both of these roles but as the project becomes larger, it is less likely that one person will be able to fill them. As mentioned, system developers are concerned about how the whole system operates. They may be concerned with a particular subsystem, until they are convinced that it is operating properly, then they will turn their attention to another subsystem. Model developers are in the best position to determine what signals are truly required to be the inputs and outputs of the model. Model designers can also make the best decision about what internal values may be of interest to future users given the current implementation of the model. To fully implement encapsulation, the model developer must also ensure that other models do not rely on values that are not part of the model's normal interface.

The data management system of ASSET recognizes the interplay between the two roles: the system developer who needs signals from all over the system and the model developer who needs models with the smallest interface possible. ASSET's data management system provides a structure such that each designer can effectively perform his or her job.

Overview

ASSET's data management system separates the data generators from the data consumers. Data generators are usually models and data consumers are called DataCollectors. The basic design is that models write data into a central repository, called the DataStore, which stores a time-history of the data. Once the data is in this repository, the DataCollectors can then pull data out. Figure 1 illustrates the relationship between Models, DataStore, and DataCollectors. This data flow is strictly in one direction. The data management system does not provide any facility to retrieve data from either DataCollectors or the DataStore and insert this data into a model.

The design of the data management system supports the model developer and system developer roles, previously described. Since model designers are in the best position to determine what values are of interest in the model, they decide what data the model can send to the DataStore. System designers decide if that model should send any data to the DataStore. In other words, those who use the model (the system designers) decide if the data from that model is of any interest.

The data management system also enjoys the benefits of encapsulation without incurring its penalties. Since there is no ability to feed data back into the models, model designers are assured that signals cannot be incorrectly injected into their models. Signals must come in to models through their input vector. In addition, model designers do not need to unnecessarily expose their implementation to allow system designers to examine the values they find interesting. The model designers also do not need to be concerned with who might want the data from their models and where to send that data. They send the data from their model to the central collection point, where the system designer can view it. The data management system provides a small "back door" through this strict encapsulation interface to allow system designers to gather the information necessary to do their job.

In addition to supporting the needs of the two types of ASSET designers, a few other interesting features come out of its design. Since all data in an ASSET simulation is stored in the same format in the DataStore, creating a structure that allows data presentation in any format is almost trivial. Through the object-oriented concept of inheritance, all DataCollectors share the same software that manages a group of channels and accesses the DataStore. If a user requires a new data format they can derive from the general DataCollector hierarchy. In this way, they only need to be concerned with the specific requirements of the data format, not the details of pulling data out of the DataStore. The concept of a data format is quite broad; in the small sense, it could mean a tab-delimited text format or a binary format, but it could also mean a graphical strip chart system or even a graphical flight display.

Also, the creation of a central repository allows the DataCollectors to be generic. The DataCollectors do not have to be aware of the interface of every model whose data they use. They only need to know what data they want, not which model generated it. The separation between models and DataCollectors goes even further. Models can run asynchronously from the DataCollectors. Synchronizing the data is accomplished in the common DataCollector software. This allows the data recording to proceed when the main computation is idle or if an ASSET simulation was running on a multiprocessor computer, on another processor.

The three major components of ASSET's data management system are the models, which generate the data, the DataStore, which serves as the central repository, and the DataCollectors, which display the data in some useful format. Each of these components is described in its own section below.

Models and Their Data

One important concept is the balance between the model designers who are interested in being insulated from the context in which their model operates and the system designers who are interested in getting all the data needed to do their job.

Since the modeler cannot know what signals the system designer will be interested in, the design of ASSET allows the modeler to specify all signals that may be of interest. These signals may include inputs, outputs, parameters, states (continuous and/or discrete) or intermediate computations. Once the modelers complete this specification they no longer need to be concerned with data management. The Model superclass [2] and the DataStore perform all necessary operations to actually get the data out of the model. Specifically, the Model superclass determines when the signals in the model are ready to be sampled. The DataStore maintains a time-history of each signal.

System designers are also aided by the design of the data management system for ASSET. These developers only need to indicate that they are interested in values from this model. If they are not interested in values from this model, then no computational penalty is incurred. If they are interested in values from this model, then they automatically get access to all values that the modeler specified. System designers are not required to view these signals if they do not want to, but they are available. System designers also have the ability to set the sampling rate. This is an integer representing a sub-division of the simulation frame rate. If the sampling rate is 3, then the data will be sampled every third frame. The DataStore has no knowledge of how often models write data into the DataStore. Since the DataStore has no concept of this sampling rate, the sampling rate can be changed while the simulation operates, including being set to zero which turns sampling off.

A class indicates that it is able to write data into the DataStore by implementing the Recordable interface[†]. The Model superclass implements this interface; therefore all ASSET models can record data in the DataStore. The Recordable interface contains two methods. The `setu-`

[†] An interface is a special Java construct, similar to an abstract class, which only contains method interfaces but not their implementations [1].

```
public void setupRecordable(int objectID) {
   datastore.addChannel(objectID, “speed”, “ft/s”);
   datastore.addChannel(objectID, “alt”, “ft”);
   datastore.addChannel(objectID, “alpha”, “deg”);
   datastore.addChannel(objectID, “beta”, “deg”);

   super.setupRecordable(objectID);
   }

public void record(int channelID, double[] values) {
   values[channelID++] = speed;
   values[channelID++] = altitude;
   values[channelID++] = alpha;
   values[channelID++] = beta;

   super.record(channelID, values);
   }
```

Figure 2 – Sample Recordable Methods.

`pRecordable()` method performs setup operations for a channel such as specifying a name and providing suggested units for the DataCollectors to use. The `record()` method actually copies the data into the DataStore. One method could be used for both functions; however, that would incur a greater performance cost. With the current structure, only the `record()` method needs to be called during the main execution loop of the simulation. An example of the two methods of the Recordable interface is given in figure 2.

The Model superclass tells the DataStore when the signals in the model are ready to be recorded. This operation involves calling the DataStore’s `sample()` method. The Model gives the value of its simulation time to DataStore with this method. The DataStore will then invoke the model’s `record()` method. Through this method, the DataStore will pass the Recordable an array. The Recordable must then fill this array with the appropriate values going to the DataStore. When the `record()` method completes, the DataStore can alert any collectors that new data has arrived.

ASSET models are the major implementers of the Recordable interface; however, any class that implements the Recordable interface can add data to the DataStore. An example of a class that may be a Recordable but is not a model would be the interface software for a cockpit simulator. The system designer may want to know exactly what is coming out of the hardware, before any modeling software uses the data. Currently in ASSET, Models are the only classes that implement the Recordable interface; so for the purposes of this paper, the terms model and Recordable are interchangeable.

DataStore

The DataStore is built as a series of channels, where each channel is a structure that contains a time history of a particular data value. At a particular instant in time, models will generate the data value and notify the DataStore. The DataStore will retain the time history of this quantity, thus creating a channel. By maintaining a history of each data value, the DataStore effectively decouples the data collection from the data output—allowing models to execute at a different rate from the DataCollectors.

The DataStore serves as the central repository for data in an ASSET simulation. Models write their data into the DataStore, and DataCollectors read data out of the DataStore and present this data to the user. This design has several features. The models do not need to make their interface available to every model in the simulation—they only need to make their interface available to those models that truly interact with the model. The data entry portion of DataStore is performance sensitive, but the DataCollectors are not; therefore, the data entry software can be tuned for a real-time environment. This partitioning allows the main simulation to operate independently of the data collection.

At its core, the DataStore is simply a three-dimensional block of `double`s, with the dimensions of models, quantities, and time. (All quantities in ASSET are stored as a `double`, a data type in Java indicating a double precision floating point value.) Since the purpose of the DataStore is to store channels (and a channel is the time history of a quantity), the dimensions of *quantity* and *time* fall out naturally. The dimension of *models* is needed because some information is the same for all channels within a model. The most prominent member of this group is the value of time. The DataStore stores a separate time sequence for each model. This allows the models to operate at different rates and even asynchronously from each other. Furthermore, since a value of time is stored for each dimension, the models can change their update rate, at any time without affecting the DataStore. The DataStore does not provide any correlation of the signals from different models—this is the job of the DataCollectors.

At construction-time the user specifies the size of the DataStore. This size is the approximate number of Java `double`s that the DataStore allocates. Nominally, the DataStore will divide this space equally among the channels. For a simulation of a given complexity, the size of the buffer indicates how many past data values the DataStore can hold; the greater the size, the longer the time history can be stored for each channel. The DataStore cannot derive a time (how many seconds of data can be held) from this parameter, because then the DataStore would know how often a model is going to write data into the DataStore. The rate at which a model writes data is entirely up to the model itself. The model can change this rate (and even stop it) at any time. Once the buffer within the DataStore is exhausted, the DataStore will write over the oldest data. The DataCollectors must ensure that they acquire the data they need before the data is overwritten. Since the DataStore operates in the main simulation loop and allocating memory[‡] during this time would probably preclude the use of ASSET in a real-time environment, the DataStore does not allocate any memory when running in the main simulation loop.

If one model's iteration rate is twice as fast as other models in the system, then this model will need twice as much storage as other models to be able to store a time-history of the same length. It is planned that the user will be able to weight the data allocation within the DataStore to account for this situation. Currently this feature is not implemented.

When the DataStore has all the data for a particular model at a particular time, it reports the arrival of this new data to the DataCollectors. The DataStore only alerts the DataCollectors; it does not send them any data. It is the DataCollectors' responsibility to determine if they are interested in the data and to read the data from the DataStore. The DataStore can alert the DataCollectors in one of two ways: as an inline method or through a Java event. Currently only the inline method is implemented. The inline technique assumes a single thread of control will flow from a model generating data, to the DataStore recording this data, and then to the DataCollectors displaying this data. The event technique provides a mechanism for the models and DataCollectors to operate independently of each other. One thread (associated with a model) can write data into the DataStore and not interfere with another thread (associated with a DataCollector) reading data out of the DataStore.

The DataStore is designed to operate as independently as possible from both models and DataCollectors. One of these components can change without forcing changes on the other components. As a result the DataStore deliberately does not understand certain concepts. For instance, the DataStore doesn't have an independent concept of time. Specifically, it doesn't assume it knows what the "next" time will be. This allows the model to change the data update rate without consequence to the DataStore. Since the DataStore cannot make any assumptions about time, the DataStore does not perform any synchronization of data between different models. The DataCollectors are responsible for correlating the data coming out of the DataStore.

DataCollectors

Once the data is in the DataStore, the data output problem becomes significantly easier; all data is available in a central location and the data can be accessed through a single interface. The real problem for the design of the DataCollectors is how to develop a structure that facilitates new data output capabilities with a minimal amount of work.

The DataCollectors are the classes that take data out of the DataStore and present this data in some format useful to the user. DataCollector is an abstract class; it only provides a structure to copy data out of the DataStore, not a full implementation. Since all the data in an ASSET simulation is stored in a common format in the DataStore, only one software module needs to be written to take data out of the DataStore; this software is the DataCollector superclass. The DataCollector superclass does not display any data itself. To display data, a class must be built that inherits from DataCollector. ASSET provides several classes to display the data including space-

[‡] Allocating memory often involves a non-deterministic execution time. Thus, its use during a real-time simulation must be avoided.

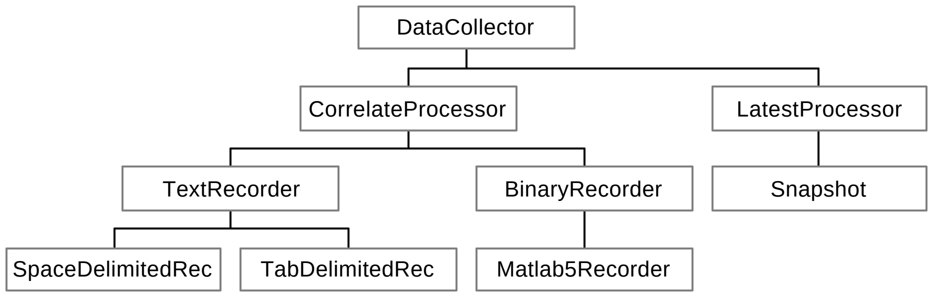

Figure 3 - DataCollector Inheritance Hierarchy

and tab- delimited recorders, MATLAB[§] recorders, and snapshots (classes that report the latest value for a channel). In the future, ASSET will provide advanced DataCollectors like graphical strip-chart plotters and graphical flight displays. The hierarchy of DataCollectors was designed to allow the user to quickly produce new software to display data in different formats.

Currently ASSET provides two classes that derive from DataCollector: CorrelateProcessor and LatestProcessor. Whereas the DataCollector superclass only understands how to take data out of the DataStore, these processor classes determine which data is processed, but they do not define how to display the data. Recognizing that models can write data at different rates, the CorrelateProcessor can correlate information between different models. The CorrelateProcessor maintains a time independent of the simulation time. It will process the data after it correlates the data from all channels for this time. The CorrelateProcessor will perform a zero-order hold on any data that has not been updated for the current time. The LatestProcessor simply creates a data set of the latest available data for all channels. Neither processor will interpolate under any circumstance. A special InterpolateProcessor is planned for such applications.

The differences between the CorrelateProcessor and the LatestProcessor can be subtle. The major difference is when a CorrelateProcessor produces a data set for time *t*, the data within this set is valid for time *t*. The LatestProcessor does not perform this correlation. If two models operate asynchronously from each other, they could have different simulation times. The LatestProcessor does not take this difference into account. Another difference between these two processors is the CorrelateProcessor can operate without intervention from the user. Once all data is available for a particular time, the CorrelateProcessor will form the data set and display the data. The user must explicitly invoke the LatestProcessor when the data is to be processed.

As discussed above, the DataStore can alert a DataCollector in one of two ways: inline and event. All DataCollectors can be alerted by either method. The inline method minimizes the amount of storage the DataStore needs to allocate and the event method minimizes the time spent in the main computational loop. With the inline method, the DataCollectors always have the data they need to display; so there is no risk that any data will be lost. Since data cannot be lost, past values do not need to be stored in the DataStore; therefore, the inline method minimizes the amount of storage. Unfortunately, this method executes data display operations during the main execution loop. These operations may involve writing to files or drawing graphics, which could have a large and nondeterministic execution time. In a real-time environment this is unacceptable. The event method minimizes the time spent in the main computational loop, but this method requires enough storage in the DataStore to buffer the data to mitigate the possibility of losing data. Using this mechanism, the when the DataStore receives new data, it simply sends a Java event to the DataCollectors and immediately returns to the main simulation loop. The DataCollectors must then access the DataStore and read out any needed data before it is overwritten. This allows the simulation to operate in real time, but the data recording, since it involves writing to the disk, can occur when the real-time computation is idle. In summary, the inline method works best for batch simulations (simulations that are not sensitive to nondeterministic execution times) and the event method works best for real-time simulations.

The table below contains examples of different processors and alerting mechanisms and examples of their use.

[§] MATLAB® is a commercial software package for computation, visualization, and programming produced by The MathWorks, Incorporated. A MATLAB recorder is an ASSET class that writes data in MATLAB's native format.

Processor	Inline Alerting	Event Alerting
Correlate-Processor	• Batch recorder • Batch plotter	• Real-time recorder • Graphical strip chart recorder
Latest-Processor	• Real-time hardware output • Batch snapshot	• Graphical data viewer • Real-time snapshot

An inheritance hierarchy for members of the DataCollector family is shown in figure 3. The leaves of this tree are classes that can be instantiated; the other members are abstract classes. A recorder is a DataCollector that writes time-history data out to a disk file. There are two types of recorders: TextRecorder and BinaryRecorder. As their names suggest, TextRecorder writes text files and BinaryRecorder writes binary files. These classes perform any necessary operations to create and manage disk files, but they do not contain any information about the format of the file. The two subclasses of TextRecorder are the space- and tab- delimited recorders. These DataCollectors write information in a simple text format with the columns of data separated by either spaces or tabs. The Matlab5Recorder will write data in a format compatible with MATLAB version 5.0 from The MathWorks, Incorporated [4]. These three classes (TabDelimitedRecorder, SpaceDelimitedRecorder, and Matlab5Recorder) essentially only contain one method to write the file in the appropriate format. If the user wanted to add a different format, they would simply inherit from either the TextRecorder or BinaryRecorder classes and define this method. A Snapshot is a DataCollector that saves the latest values of all the channels to a file. This is useful for capturing a view of the model at a particular instant of time (that is, a "snapshot").

The main configuration parameter for recorders is the recording rate. Other configuration parameters include the selection of channels, their associated units, and formatting information such as the number of decimal digits to display. The recording rate indicates how often the collector will record the data to a file. This feature allows the user to distinguish between the recording rate and the simulation rate. The simulation rate may increase or decrease, but if the recording rate does not change, the amount of data in the file will not change.

Just as the DataStore does not assume any rate that data arrives, the DataCollectors also do not assume any arrival rate for data. Generally, DataCollectors do not store any data themselves; they rely on DataStore to retain past values for each channel. The user must be careful about relying on the DataStore to store past values. When the DataStore fills all its storage for a channel, it will overwrite the oldest values. The CorrelateProcessor will detect this condition (this condition has no meaning to a LatestProcessor). To avoid this situation, the user can either use the inline method to alert the collector—incurring any resulting performance penalties—or ensure the DataStore has enough memory.

The user may add any number of channels to a DataCollector from the DataStore. There is no restriction on what data any DataCollector may receive; if the data is in the DataStore, all DataCollectors can read it. Several DataCollectors can simultaneously read the same channel without any difficulty. When the user adds a channel to the collector, one important specification is the unit to be used for displaying the quantities recorded on this channel. Since all data in the DataStore is specified in internal units [2], the quantity needs to be converted to the desired unit prior to outputting the data. If the user of a DataCollector does not specify the units for a channel; then suggested units provided by the model writers will be associated with the channel.

Status

Most of the software described in this paper for the ASSET's data management system has been in service for approximately nine months. The software to implement the event-oriented method for alerting DataCollectors has not been implemented. Using a representative simulation model (a version of the F-16 aircraft model described in [5]), the data management system (with the Matlab5Recorder) was able to record 210 signals in 35 microseconds on a 550 MHz Pentium III machine. This time just measures the performance impact of the data management system within the main computational loop of a simulation (i.e., the time to record the data into the DataStore). This time does not include much of the computational load of a DataCollector, including any time to actually write data to disk.

The software has demonstrated that new data formats can be added quickly. The authors were able to build the SnapShot collector in approximately one hour. It is expected that less experienced developers would take longer, but not excessively longer, to develop new collectors.

In addition to completing the event alerting software, future enhancements to this software will include testing this in a real-time environment. Full testing of the real-time performance of this system will require the event alerting mechanism to be completely functional. Also the software must be verified to work within a multi-processing computer system. Achieving high performance while maintaining data integrity will be a challenge in this type of environment. The software was designed to allow multiple independent threads to write into the DataStore simultaneously; however, this operation will need to be verified.

Summary

ASSET's data management system provides a solution to a fundamental problem in simulation software development. Software can be developed faster and with higher quality if the modeler can be insulated from other software in the system. However, this model encapsulation also hinders legitimate use by others who require access to all signals so they can gain an understanding of the system as a whole. The data management system, employing a central repository, combines encapsulation with one-way visibility to meet both requirements. As a side benefit of providing a central repository for the data, new data formats can be added with very little effort.

References

[1] Arnold, Ken; Gosling, James. *The Java Programming Language, Second Edition.* Addison Wesley Publishing Company, Reading, MA, 1998.

[2] Derry, Stephen; Maddalon, Jeffrey. *Implementing Dynamic System Models in the ASSET Simulation Framework*. AIAA 2000-4393. Modeling and Simulation Technology Conference, Denver, CO, August 2000.

[3] Booch, Grady. *Object-Oriented Analysis and Design with Applications, Second Edition*. Benjamin / Cummings Publishing, Redwood City, CA, 1994.

[4] The MathWorks, Inc. *MAT-File Format, Version 5.* The MathWorks, Inc. © 1999. http://www.mathworks.com

[5] Stevens, Brian L.; Lewis, Frank L. *Aircraft Control and Simulation.* John Wiley and Sons, Inc., New York, NY, 1992.